Characterizing the Branch Misprediction Penalty

Stijn Eyerman[†] James E. Smith[‡] Lieven Eeckhout[†]

[†]ELIS, Ghent University, Belgium
[‡]ECE, University of Wisconsin–Madison, USA
{seyerman,leeckhou}@elis.UGent.be, jes@ece.wisc.edu

Abstract

Despite years of study, branch mispredictions remain as a significant performance impediment in pipelined superscalar processors. In general, the branch misprediction penalty can be substantially larger than the frontend pipeline length (which is often equated with the misprediction penalty). We identify and quantify five contributors to the branch misprediction penalty: (i) the frontend pipeline length, (ii) the number of instructions since the last miss event (branch misprediction, I-cache miss, long D-cache miss)—this is related to the burstiness of miss events, (iii) the inherent ILP of the program, (iv) the functional unit latencies, and (v) the number of short (L1) D-cache misses. The characterizations done in this paper are driven by 'interval analysis', an analytical approach that models superscalar processor performance as a sequence of inter-miss intervals.

1 Introduction

Branch mispredictions are a significant impediment to performance, especially in deeply pipelined processors. The total performance penalty due to branch mispredictions is the product of the branch misprediction rate, *i.e.*, the fraction of mispredicted branches, and the branch misprediction penalty, *i.e.*, the number of lost execution cycles per mispredicted branch. The penalty for a branch misprediction can be significantly larger than the frontend pipeline length (*i.e.*, the pipeline refill time). Figure 1 shows the average branch misprediction penalty for the SPEC CPU2000 integer benchmarks—details regarding the experimental setup are given later. In these experiments the front-end pipeline is set at five pipeline stages. As the graph indicates, the branch misprediction penalty (measured in clock cycles) is always larger than the time it takes to traverse the front-end pipeline length. For some benchmarks the penalty observed on a branch misprediction is several times the front-end pipeline length.

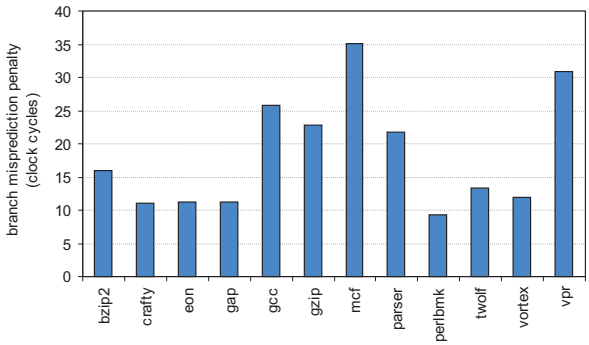

Figure 1. The branch misprediction penalty for the SPEC CPU2000 integer benchmarks.

There are a number of contributors to the branch misprediction penalty, of which the pipeline re-fill time is only one. In this paper, we study the contributors to the branch misprediction penalty in detail and analyze them in terms of program characteristics. We show that the performance penalty per branch misprediction is a function of:

1. The frontend pipeline length;
2. The number of dynamic instructions since the last significant miss event (branch misprediction, I-cache miss, long data cache miss (*e.g.*, an L2 miss))—this is a function of the program's locality and predictability;
3. The average critical dependence path, *i.e.*, for a given instruction window size, the length of the average instruction dependence chain. The length of the critical path can be further divided into three components:
   (a) The inherent critical path with single cycle latencies; this is the count of instructions along the critical path.
   (b) The instruction latencies; these amplify the critical path effect, and are a function of the program's instruction mix, *i.e.*, the more long latency instructions on the critical path, the longer the critical path, as measured in clock cycles.

(c) The number of short (L1) data cache misses; these further amplify the critical path length. In our analysis, L1 data cache misses are modeled separately from the other types of miss events because, in general, their effect on performance is different. With adequate load/store queuing, L1 data cache miss latencies can be successfully overlapped with other instructions, unlike the other types of miss events.

This paper also demonstrates an application of *interval analysis* – a general technique for modeling superscalar processors. In interval analysis, superscalar processor performance is viewed as a sequence of inter-miss intervals. The misses that define the intervals are branch mispredictions, (L1 and L2) I-cache misses and long (L2) D-cache misses. Interval analysis allows sections of dynamic program execution to be studied more-or-less in isolation; we will use it to isolate mispredicted branch behavior.

2 Experimental setup

2.1 Processor model

We consider superscalar processors as illustrated in Figure 2. This is a more-or-less generic processor, similar in style to many processors in use today. The processor, as drawn, incorporates a number of design parameters and program-related characteristics that affect the overall performance. A key parameter, often referred to as the "width" of the superscalar processor, is denoted as I in Figure 2. The width defines the number of instructions that each pipeline stage can process per cycle, as well as the rate at which instruction issue, execution, and commit can be sustained.

At the left side of the figure is the instruction delivery subsystem which combines elements of instruction fetch, instruction buffering, and branch prediction. It is important that the instruction delivery subsystem provides a sustained flow of instructions that matches the capacity of the rest of the processor to issue, execute, and commit instructions. Because of the variability in dynamic basic block sizes (distances between branches), the peak fetch width, parameter F in Figure 2, will typically be larger than the pipeline width, with an instruction fetch buffer to moderate the flow into the pipeline.

After instructions are fetched, they are decoded, their registers are renamed, and they are dispatched into an instruction issue buffer and a reorder buffer (ROB). In a modern superscalar processor, the instruction issue buffer is separate from the ROB. The role of the ROB is to maintain the architected instruction ordering so that the correct process state can be restored in the event of a trap or a branch misprediction.

benchmark	input	simulation point
bzip2	program	10
crafty	ref	1
eon	rushmeier	19
gap	ref	2,095
gcc	166	100
gzip	graphic	4
mcf	ref	317
parser	ref	17
perlbmk	makerand	2
twolf	ref	32
vortex	ref2	58
vpr	route	72

Table 1. The benchmarks used in this paper along with their inputs and simulation points.

In this study, we assume that the superscalar processor is *balanced* with respect to its given width and the programs that will be run on it. This means that the issue buffer and ROB sizes are adequate to achieve the maximum issue rate in the absence of significant miss events such as branch mispredictions, I-cache misses and long (L2) D-cache miss events. Short (L1) D-cache miss events are treated differently since L1 D-cache misses can be hidden through the out-of-order execution of independent instructions [2]. Adequate load/store buffering and a sufficiently large instruction window can hide most of the L1 D-cache miss penalties with almost no performance loss. This requires the ROB, the issue buffer and related structures to be sized appropriately in a balanced design.

When significant miss events (such as branch mispredictions, I-cache misses and long D-cache misses) are considered, the maximum issue rate can no longer be achieved all (or most) of the time, but intuitively, balance means that the maximum issue rate can still be achieved at least some of the time[1]. The assumption of a balanced design also assumes adequate fetch resources to provide sustained instruction delivery at the rate equal to the issue width.

2.2 Simulation infrastructure and method

All simulation results are measured using a modified version of SimpleScalar/Alpha v3.0. We use all the integer SPEC CPU2000 benchmarks; we do not consider the floating-point benchmarks because these benchmarks suffer less from branch mispredictions. In order to limit the total simulation time in our experiments we use the (single) 100M simulation points provided by SimPoint [5]. Table 1 shows the benchmarks and their reference input. The binaries are taken from the SimpleScalar website[2]. The superscalar processor model that we assume in this study is tabulated in Table 2. All results presented in this paper are based on this processor model unless mentioned otherwise.

[1]Note that we do not provide an exact number for what 'some of the time' means. This is up to the designer, but is probably at least 5 percent.

[2]`http://www.simplescalar.com`

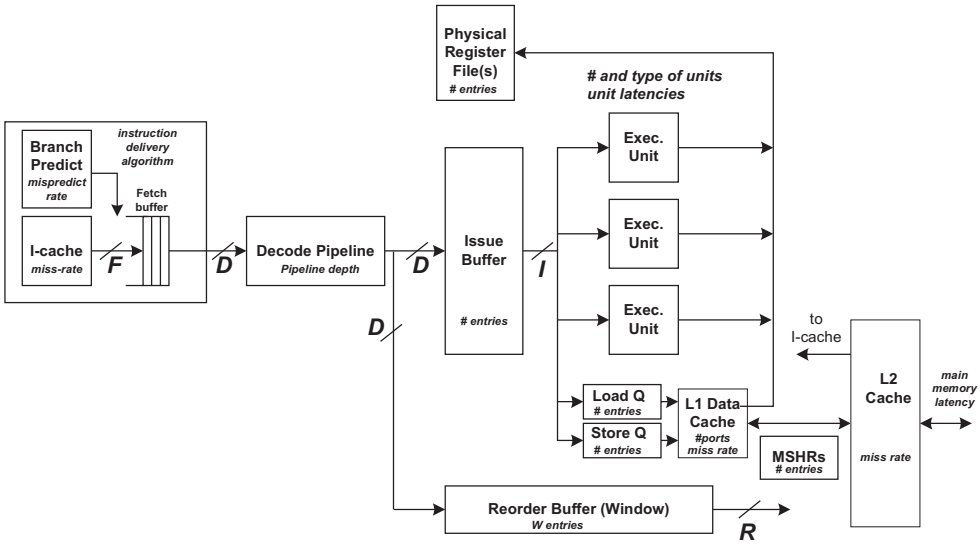

Figure 2. A superscalar processor.

ROB	64 entries
processor width	$D = I = R = 4$ wide
fetch width F	8 wide
latencies	load 2 cycles, mul 3 cycles, div 20 cycles, arith/log 1 cycle
L1 I-cache	8KB direct-mapped, 32-byte cache lines
L1 D-cache	16KB 4-way set-associative, 32-byte cache lines
L2 cache	unified, 1MB 8-way set-associative, 128-byte cache lines 12 cycle access time
main memory	200 cycle access time
branch predictor	hybrid predictor consisting of 4K-entry meta, bimodal and gshare predictors
front-end pipeline	5 stages

Table 2. Processor model assumed in our experimental setup.

3 Interval analysis

We will study the effects of branch mispredictions using interval analysis, a method which gives insight into the interactions in a superscalar processor without the detail of tracking individual instructions. With interval analysis, execution time is broken into discrete intervals by disruptive miss events (such as cache misses and branch mispredictions). Then, statistical processor and program behavior allows us to characterize superscalar behavior for each interval type.

The basis for the model is that a superscalar processor is designed to stream instructions through its various pipelines and functional units, and under optimal conditions, it sustains a level of performance equal to its issue (or commit) width. However, the smooth flow of instructions is often disrupted by miss events such as cache misses and branch mispredictions. When a miss event occurs, the issuing of useful instructions eventually stops; there is then a period when no useful instructions are issued until the miss event is resolved and instructions can once again begin flowing.

This interval behavior is illustrated in Figure 3. The number of instructions committed per cycle (IPC) is shown on the Y axis and time (in clock cycles) is on the X axis. As illustrated in the figure, the effects of miss events divide execution time into intervals. Intervals begin and end at the points where instructions just begin issuing and committing following recovery from the preceding miss event. That is, the interval includes the time period where no instructions are issued following a particular miss event. We define interval execution time as the number of clock cycles to execute an interval and we define interval length as the number of (correct-path) instructions executed in the interval.

By dividing execution time into intervals, we can then analyze the performance behavior of the intervals individually. In particular, we can, based on the type of interval (the miss event that terminates it), describe and evaluate the key performance characteristics.

An incomplete version of interval analysis for studying instruction delivery was proposed by Michaud *et al.* [3, 4]. A more complete method for interval analysis, extended to other types of miss events, was proposed by Taha and Wills [6]; in that work, penalties due to miss events were generated experimentally. Karkhanis and Smith [2] propose analytical models for various types of miss event behavior and validate them experimentally. However, their models are applied in a dual approach to interval analysis, which we call *gap analysis* here. Gap analysis assumes that steady-state performance is sustained most of the time. When a miss event occurs, performance falls to zero; when the miss is resolved, performance ramps up again to steady-state. The result is a gap in the steady-state performance. Interval analysis and gap analysis might appear to be equivalent. However, there is one subtle but important difference. In cases where miss events occur close together (in bursts),

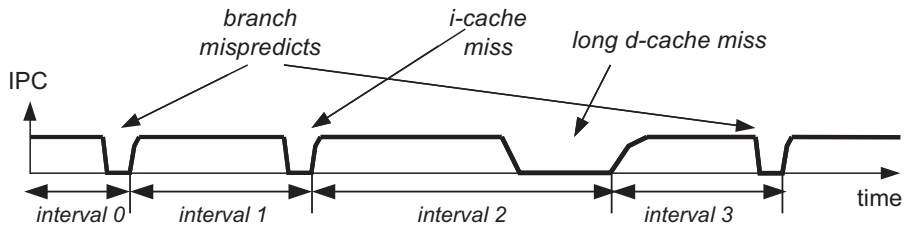

Figure 3. Basic idea of interval analysis: performance can be analyzed by dividing time into intervals between miss events.

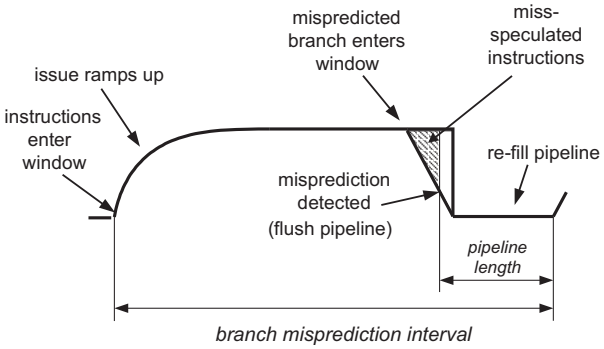

Figure 4. Interval behavior for a branch misprediction.

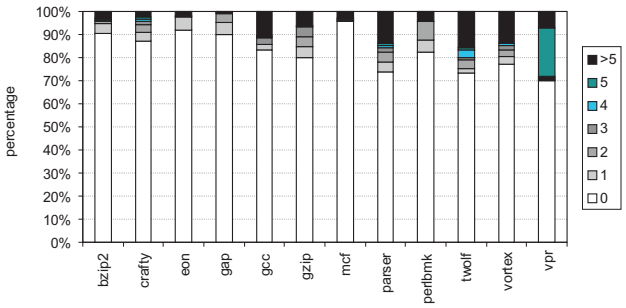

Figure 5. Difference in number of cycles between window drain time and branch resolution time.

steady-state performance is never achieved. These situations are handled more naturally through interval analysis.

3.1 Branch misprediction intervals

Figure 4 shows the timing for a branch misprediction interval. This graph plots the number of instructions issued (Y-dimension) versus time (X-dimension); in a sense, this is average or typical behavior and the plot has been smoothed for clarity. At the beginning of the interval, instructions begin to fill the window and instruction issue ramps up. Then, at some point, the mispredicted branch enters the window. At that point, the window begins to drain good instructions (*i.e.*, those that will eventually commit). Here, the emphasis is on good instructions, because miss-speculated instructions following the mispredicted branch will continue filling the window, but they will not contribute to the issuing of good instructions. Nor, generally speaking, will they inhibit the issuing of good instructions if it is assumed that the oldest ready instructions are allowed to issue first.

Eventually, the mispredicted branch is discovered. An important observation is that this very often coincides with the end of the window drain; *i.e.*, the mispredicted branch instruction is one of the very last good instructions to be executed. This observation is supported by the data in Figure 5, collected for the SPEC benchmarks. This graph shows the difference between the window drain time and the branch resolution time measured in number of cycles, given as a percentage of the number of branch mispredictions. We observe that between 70% and 95% of the time, the mispredicted branch instruction is issued during the same cycle as the issue buffer becomes drained. The only exception is benchmark vpr where most of the time is spent in a loop with a recurrent, loop-carried dependence for which the loop-terminating branch is independent of the long dependence chain.

When the mispredicted branch is resolved, the pipeline is flushed and is re-filled with instructions from the correct path. During this re-fill time, there is a zero-issue region where no instructions issue, and, given our above observation, the zero-region is approximately equal to the time it takes to re-fill the front-end pipeline (*i.e.*, a number of clock cycles equal to the front-end pipeline length.)

For a balanced pipeline, the timing analysis for a branch misprediction interval is shown in Figure 6. Here, the total interval time has three sub-intervals (A, B, and C in the figure). The time required for the first sub-interval A is simply n/I, the total number of instructions in the interval (n) divided by the issue width (I); this is the time from the beginning of the interval until the mispredicted branch is placed (dispatched) into the window.

The second sub-interval B is the time it takes for the misprediction to be discovered after the branch is dispatched into the window. If we assume that the branch is the last good instruction to be issued and executed, then the time for the second sub-interval is approximately equal to the time it takes to empty (drain) the window of good instructions be-

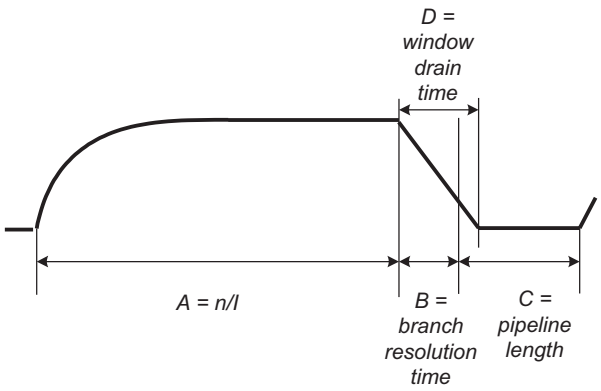

Figure 6. Timing for an interval ending in a mispredicted branch; assumes a balanced processor design.

ginning at the time the mispredicted branch enters the window, *i.e.*, B = D. The third sub-interval C is the pipeline fill time and is equal to the pipeline length.

Defining the branch misprediction penalty. A key observation regarding the first sub-interval, A, is that in a balanced processor, instructions can always be placed into the window at a sustained rate of *D* instructions per cycle. Even though instructions do not initially issue at the (maximum) issue rate *I*, as the window eventually becomes full, the issue rate will rise to *I*, so the window is emptied at the same rate it is being filled, and balance will be achieved. The point is that in a balanced processor, dispatch will never (or very rarely) block due to a full window in the absence of miss events. It then follows that the branch misprediction penalty is the difference between the time the mispredicted branch first enters the window and the time the first correct-path instruction enters the window following discovery of the misprediction, *i.e.*, B+C in terms of Figure 6.

Contributors to the branch misprediction penalty. If the number of instructions preceding the mispredicted branch is sufficiently large, then the penalty will be nearly a full window drain time (which, as we shall see, can be longer than the pipeline fill time) plus the pipeline fill time. That is, the penalty can be more than twice the pipeline fill time. On the other hand, if the interval preceding the mispredicted branch is very short, then the penalty is closer to a single pipeline fill time. This means that when there is a burst of branch mispredictions, the branch misprediction penalty will be shorter than for isolated branch mispredictions.

A second observation is that the branch misprediction penalty is a function of the program's average critical dependence path length (for a given window size). This is an important program characteristic. In general, we are interested in not only the critical path that leads to a mispredicted branch, but also the critical path length averaged over all instruction windows [4]. A program's instruction level parallelism (ILP) is inversely related to its average critical path length; *i.e.*, the longer the average critical path, the less the ILP.

The drain time is influenced by the average critical path length in two ways. First, a program with a long average critical path (low ILP) requires more instructions in the ROB to sustain the maximum issue width during steady-state. As such, when a mispredicted branch enters the window, there will be more instructions buffered in the window for a long critical path program than for a short critical path program. Obviously, draining a large number of instructions from the window takes longer than draining a small number of instructions. Then, the second drain time effect is that a program with a longer average critical path will drain the window at a slower rate, thereby increasing the branch misprediction penalty even further. This, and other, related effects are discussed in more detail in section 5.

3.2 Interactions with other miss events

Branch mispredictions do not occur in isolation; they interact with other miss events. The penalty for a particular branch misprediction often depends on the preceding miss event (and, conversely, can affect the next miss event).

From an interval analysis perspective, which focuses on the issuing of good instructions, the interaction between branch mispredictions and I-cache misses (or consecutive branch mispredictions) is limited because the penalties do not overlap. That is, branch mispredictions and I-cache misses serially disrupt the flow of good instructions so their negative effects do not overlap. Consequently, the interaction between branch mispredictions and I-cache misses is limited to the effect of the interval length that separates them. In general, as discussed above, the longer the interval preceding a branch misprediction, the larger the branch misprediction penalty.

The interactions with D-cache misses are more complex. As mentioned before, short D-cache misses are considered as long latency unit instructions in a balanced superscalar out-of-order processor design. Hence, we first focus on long D-cache misses.

When a long data cache miss occurs, *i.e.*, from the L2 to main memory, the memory delay is typically quite large—on the order of a hundred or more cycles. This delay cannot be hidden, and the penalty for a long D-cache miss will interact with, and can affect the observed branch misprediction penalties. In earlier work [1] it was found that the long miss penalty results predominantly from the following sequence of events: the load blocks at the ROB head, the ROB fills, dispatch stops due to the full ROB, and finally issue ceases.

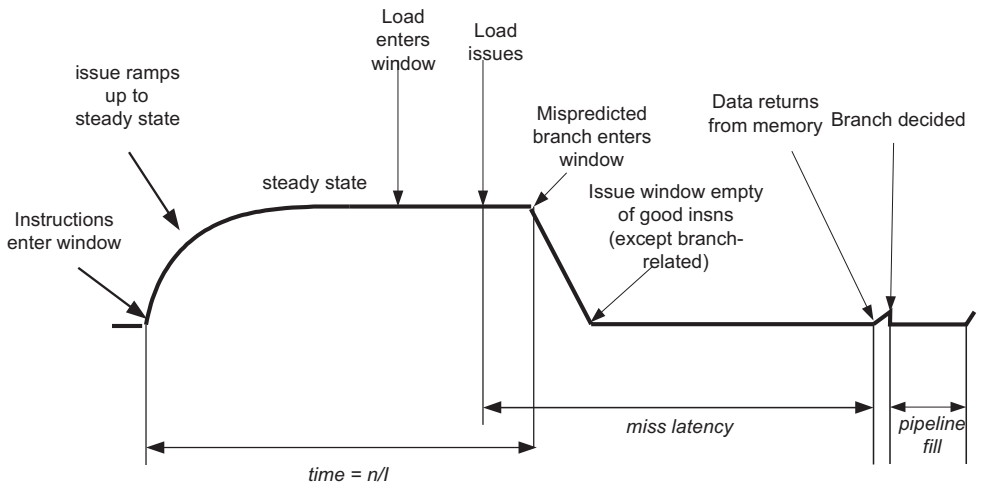

Figure 7. A long D-cache miss followed by a mispredicted branch that depends on the miss data.

Now we consider the effect on branch misprediction penalty when the mispredicted branch immediately follows a long D-cache miss. By 'immediately' we mean within the W (window/ROB-size) instructions that follow the first long D-cache miss; these instructions (including the branch) will make it into the ROB before it blocks. If the branch is not in the ROB when it fills and blocks dispatch, then the branch penalty and the long D-cache miss penalties will serialize. If the mispredicted branch does make it into the ROB, then there are two cases to consider. In one case the long D-cache miss feeds the mispredicted branch, *i.e.*, the load miss reads data on which the branch depends. In this case, the miss penalties serialize. This is illustrated in Figure 7. This case has an interval that ends with the mispredicted branch. If the branch enters the window close to the time the load issues, then the only extra latency is a pipeline fill time, which is small compared with the memory latency. In the second case, the mispredicted branch is not fed by the long D-cache miss, and in this case the misprediction penalty is hidden under the long D-cache miss penalty.

4 Critical path characterization

As observed earlier, the average critical dependence path is a program characteristic that affects the branch misprediction penalty. In this section, we will provide more detail regarding this measure, including data for benchmark programs.

We are interested in the average critical path, as a function of the instruction window size. Here we use the term window in the sense it was originally used [7], *i.e.*, it is a contiguous region of dynamic instructions, usually of some fixed size, where the first (oldest) instruction in the window is unexecuted. Other instructions in the window may or may not be executed. In terms of a superscalar processor implementation, the instructions held in the ROB correspond to instructions in the window. In contrast, the instructions in the issue buffer are the subset that have not yet issued.

The critical path metric was suggested by Michaud *et al.* [4]. First, for a given window of instructions, we use the data dependence arcs to find the critical path of instructions; *i.e.*, the longest sequence of dependent instructions. For an overall program, we take a fixed window size and 'slide' it continuously over the dynamic instruction stream, and determine the average critical path length over the program. This can be done efficiently by computing the critical path length incrementally while sliding over the dynamic instruction stream. By doing so, the algorithm also computes the critical path length simultaneously for a range of window sizes; the algorithm is based on the fact that the critical path length is always larger in a large window than in a smaller window.

We define K(W,P) to be the average critical path length for program P and window size W. If we assume all instructions have unit latency, then intuitively, the average rate at which instructions can be issued and committed for program P is $i = W/K(W,P)$. Hence, K(W,P) is a good measure of inherent ILP in program P. This measure was first proposed in [4], and we have found it to be a good one.

For the SPEC integer benchmarks, we have evaluated K(W,P). The results are plotted in Figure 8 on a log-log scale for unit execution latencies. As was observed in Michaud *et al.* [4], these curves are a straight line (at least in the region where practical superscalar processors are likely to be built). Hence, there is a power law relationship between W and K(W,P); *i.e.*, $K(W,P) \approx \alpha^{-1} \cdot W^{1/\beta}, \beta \geq 1$. The values of α and β are program dependent. In Michaud *et al.*, it is assumed that $\beta \approx 2$; however, we observe that β takes on a range of values from 1.3 to 2.4. The value of α is in the range 0.9 to 1.5 for our data, see Table 3. Generally speaking, the higher the value of β, the shorter the critical path and the more ILP is present.

	vpr	perl	parser	twolf	gzip	gap	crafty	vortex	eon	bzip2	mcf	gcc
α	1.42	1.45	1.33	1.23	1.32	1.29	1.30	1.41	1.46	1.13	1.04	0.92
β	1.32	1.37	1.48	1.49	1.51	1.55	1.60	1.69	1.70	1.76	1.85	2.40

Table 3. Power law estimate of K(W,P) as a function of α and β; benchmarks are sorted by increasing β.

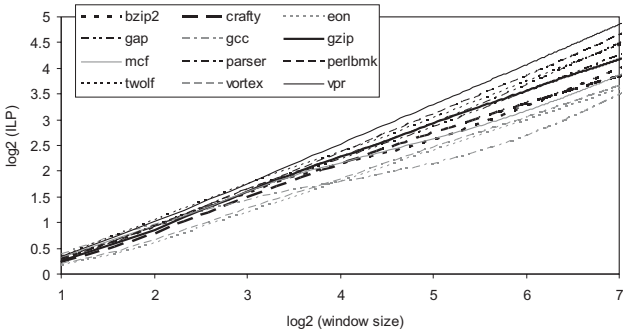

Figure 8. The average critical path length K(W,P) as a function of window size on a log-log scale.

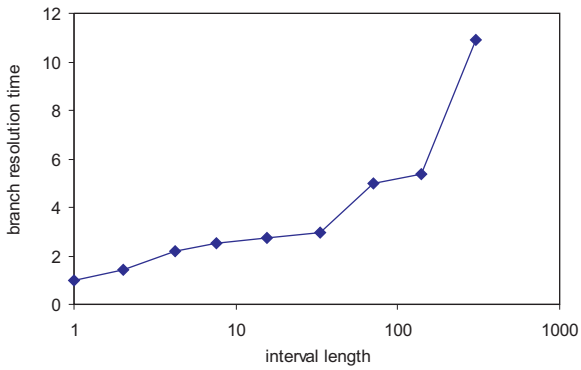

Figure 9. Branch resolution time as a function of interval length.

5 Quantifying the branch misprediction penalty

We now quantify the branch misprediction penalty. We first quantify the absolute branch misprediction penalty and subsequently discuss the relative branch misprediction penalty.

5.1 Absolute branch misprediction penalty

The absolute branch misprediction penalty is defined as the number of cycles lost due to a mispredicted branch. Referring to Figure 6, this is the branch resolution time B plus the pipeline length C. The pipeline length is fixed, and the branch resolution time determined by (i) the interval length preceding the branch misprediction, which is correlated with the burstiness of miss events, and (ii) the average critical dependence path which is a function of the inherent ILP in the program, the functional unit latencies, and the short (L1) D-cache misses. We will now quantify these two factors after which we present a stacked branch misprediction penalty model. For now, we limit ourselves to branch misses that do not overlap with long D-cache misses; overlapping misses will be treated later.

5.1.1 Impact of interval length

We first study the impact of the interval length on the branch misprediction penalty. As discussed in section 3.1 the penalty tends to be smaller in case of bursty misprediction behavior. But, note that the length of a branch misprediction interval is determined by the position of the prior miss event, regardless of the type. Hence, when we say 'burstiness' we mean the collective burstiness of miss events, not just branch mispredictions. Figure 9 quantifies the average branch resolution time (averaged over all benchmarks) as a function of interval length—we assume unit execution latency in this graph.

This graph clearly shows that the branch misprediction penalty increases for longer interval lengths. Bursty miss behavior with shorter interval lengths on the other hand, tends to yield shorter branch misprediction penalties. The reason is that if the interval contains a small number of instructions, the window contains a small number of instructions when the mispredicted branch enters the window, making window drain time smaller.

5.1.2 Impact of average critical dependence path

The second factor in the branch misprediction penalty is the average critical dependence path. We first assume unit execution latencies—this is to capture the inherent program ILP. Recall that a low-ILP program (one with a long average critical path length) is expected to fill the ROB with more instructions than a high-ILP program. The more instructions in the ROB in conjunction with the low ILP during window drain results in an overall longer drain time or branch resolution time.

Figure 10 displays a scatter plot that shows the window drain time on the Y axis as a function of the average critical path length on the X axis; again we assume unit latency in this graph. Along the X axis, the low-ILP programs are

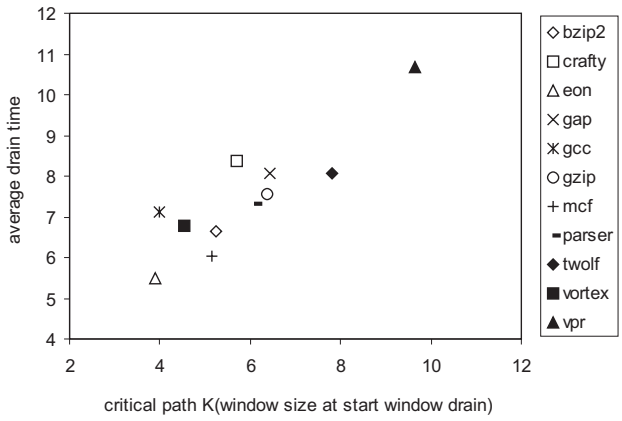

Figure 10. Scatter plot showing window drain time versus critical path length.

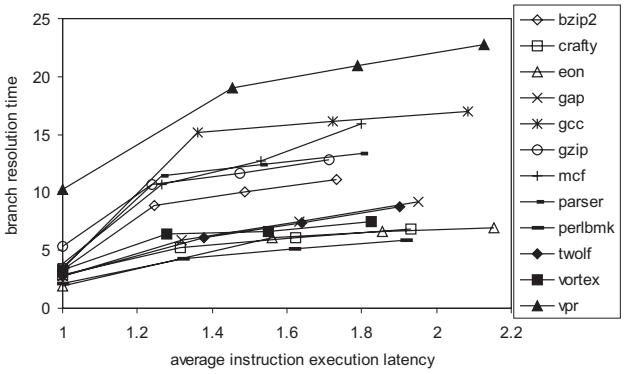

Figure 11. The impact of non-unit latency on the branch resolution time.

situated on the right hand side of the graph. The critical path on the horizontal axis is measured using the K(W,P) we described in section 4. The window size chosen for computing the critical path length is the size at the time the branch enters the window. The data shown in this graph are for relatively long interval lengths in the range [90,110]; this is to exclude the effect of short interval lengths as discussed above. The benchmark perlbmk does not appear in this graph because of the very high branch misprediction rates for this benchmark which results in very short interval lengths only – there are no interval lengths within the range [90,110]. This graph clearly shows there is a strong correlation between window drain time and inherent ILP. For example, a program like vpr has very low ILP—the β value in the power law estimation of K(W,P) for vpr equals 1.3 which is the lowest β observed among the benchmarks, see Table 3. As a result, window drain time is significantly longer than for the other benchmarks. As mentioned before, vpr spends most of its time in a loop with loop-carried dependencies, *i.e.*, the amount of ILP is limited by program parallelism rather than machine parallelism. On the other side of the spectrum we observe the following benchmarks: eon, mcf, bzip2, vortex and gcc. These benchmarks show the shortest average drain time (see the projected data points on the vertical axis). This correlates very well with the high β values reported in Table 3. These benchmarks are indeed the benchmarks with the highest inherent ILP. The remaining benchmarks show moderate average drain times; the β values for these benchmarks are around 1.5. The correlation coefficient of the data shown in Figure 10 is 84.7% (71.8% if we exclude vpr). We can thus conclude that there is indeed a strong correlation between the program's inherent ILP and the average drain time—benchmarks with low ILP tend to have a large drain time, benchmarks with high ILP tend to have a shorter drain time.

Non-unit latencies. In the above experiments we assumed unit latencies to quantify inherent ILP. However, the branch misprediction penalty is also dependent on the functional unit mix. To illustrate this, we now assume non-unit execution latencies, *i.e.*, we assume the execution latencies given in Table 2 (multiply 3 cycles and divide 20 cycles) and in addition we vary the L1 D-cache access latency from 1 to 4 cycles and measure the effect on the average branch resolution time and window drain time. The results are shown in Figure 11 in which the branch resolution time is shown as a function of the average instruction execution latency. This graph clearly demonstrates that the branch misprediction penalty increases with increased instruction execution latencies.

Short D-cache misses. Until now we assumed an ideal L1 D-cache, *i.e.*, every load from L1 is a hit. If a non-ideal L1 D-cache is assumed, the loads that hit in the L2 cache can be modeled as long latency instructions, with the execution latency being the access time to the L2 cache. When modeling the L2 hits in this way, we obtain similar results to those shown in Figure 11 (results are not explicitly given here due to space constraints). In other words, the branch misprediction penalty increases with increasing L2 cache access latencies.

Similarly, we can compute the impact of the L1 D-cache miss rate on the branch resolution time. Again, we obtain similar results to Figure 11. As such, we conclude that the branch resolution time increases with increasing L1 D-cache miss rate.

5.1.3 Stacked branch misprediction penalty model

When all the components of the branch misprediction penalty are put together we obtain the results given in Figure 12. This data is for a four-wide processor, and the stacked branch misprediction penalty model breaks the total penalty into five components: (i) the front-end pipeline

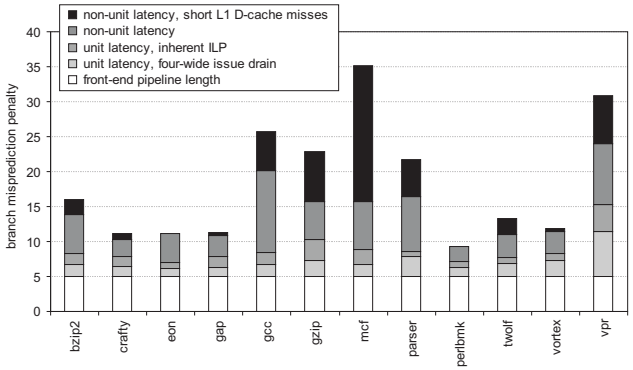

Figure 12. Stacked branch misprediction penalty model.

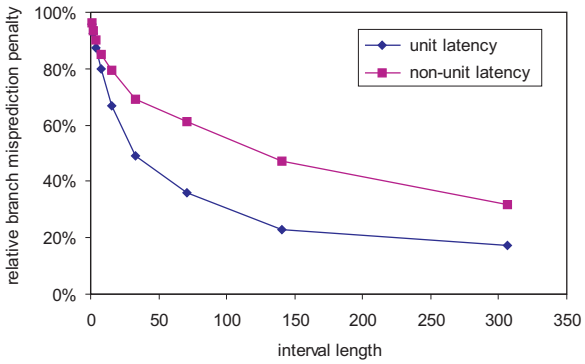

Figure 13. Relative branch misprediction penalty as a function of interval length.

fill time, (ii) the drain time, (iii) the penalty under unit latency assumptions—this quantifies the inherent ILP when draining the window, (iv) the additional penalty for non-unit latency, and (v) the additional penalty for non-ideal L1 D-cache. Note that the top line of this graph is identical to the top line in Figure 1. This graph shows that the components vary widely among the benchmarks. For example, for vpr the branch misprediction penalty increases significantly due to its low inherent ILP; the low ILP results in the window filling up which in its turn, results in a long drain time. In terms of the non-unit latencies, the branch misprediction penalty increases substantially for mcf due to its very high L1 D-cache miss rate. Also for gcc, gzip, parser and vpr, the branch misprediction penalty seems to increase significantly due to L1 D-cache misses; the reason is that the mispredicted branch is dependent on loads missing in the D-cache. For the eon and perlbmk benchmarks on the other hand, the branch misprediction penalty increases more from non-unit latencies than from D-cache misses. This suggests that there are no D-cache misses on the critical path leading to the mispredicted branch for these benchmarks.

5.2 Relative branch misprediction penalty

The above sections discussed the absolute branch misprediction penalty. Now we discuss the relative branch misprediction penalty, which is defined as the absolute branch misprediction penalty divided by the total time spent within the interval that ends with a mispredition. In other words, referring to Figure 6, we quantify the fraction of penalty time B+C compared to the total interval execution time A+B+C. Figure 13 quantifies the relative branch misprediction penalty as a function of interval length.

As seen from this graph, the relative penalty decreases with increasing interval length. In fact, for small interval lengths, most of the time is spent recovering from the mispredicted branch. Because the absolute penalty is smaller for shorter interval lengths (see Figure 9), the relative penalty is higher when the absolute penalty is lower. Note that even for fairly long intervals, the relative branch misprediction penalty is still fairly large, *e.g.*, an interval length of 300 instructions has a 20% relative branch misprediction penalty in a unit latency experiment, and a 31% relative branch misprediction latency in a non-unit latency experiment, see Figure 13. Note also that the relative penalty increases with increasing instruction execution latencies. This is explained by the increasing drain time penalty with an increasing instruction latency.

6 Classifying branch mispredictions

As mentioned before, the branch misprediction penalty depends on the interaction between the branch misprediction and other miss events. We now classifiy branch mispredictions based on the preceding miss event's type. We can make the following classification: (i) a branch misprediction follows a branch misprediction [br-br], (ii) a branch misprediction follows an L1 I-cache miss [br-il1], (iii) a branch misprediction follows an L2 I-cache miss [br-il2], (iv) a branch misprediction follows an L2 D-cache miss [br-dl2]. The latter case can be further subdivided depending on whether (v) the branch comes more than ROB-size instructions after the L2 D-cache miss [br-dl2, >W], or (vi) the branch comes less than ROB-size instructions after the L2 D-cache miss [br-dl2, <W]. Once more, (vi) can be further subdivided into (vii) the branch depends on the L2 D-cache miss, and thus the penalties serialize [br-dl2, <W, dep], and (viii) the branch is independent on the L2 D-cache miss, and thus the branch penalty is hidden under the L2 D-cache miss [br-dl2, <W, indep].

Figure 14 shows fractions of branches according to this classification. We observe that for the majority of all the benchmarks, a branch misprediction is usually preceded by a branch misprediction or an L1 I-cache miss. Very few branch mispredictions are preceded by an L2 I-cache miss

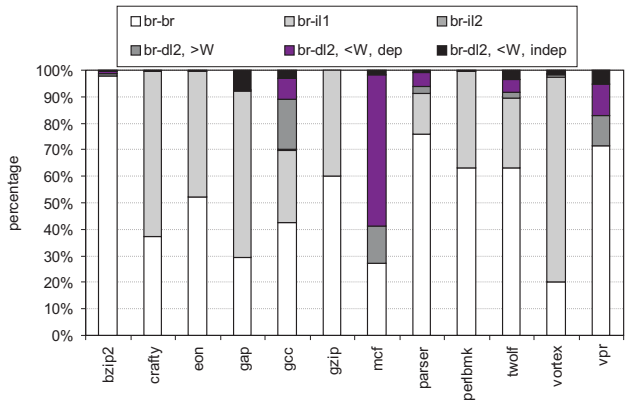

Figure 14. Classifying branch mispredictions based on their interaction with other miss events.

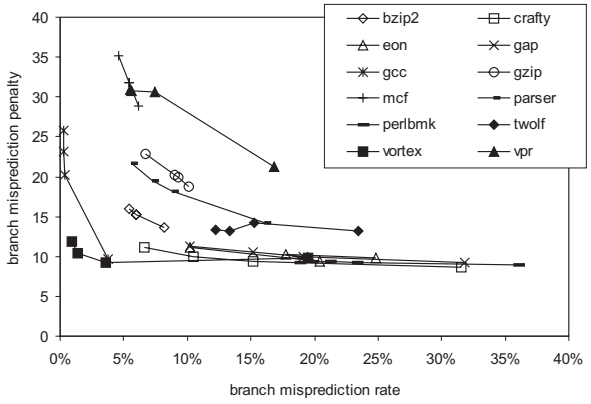

Figure 15. The branch misprediction penalty as a function of the branch misprediction rate.

because L2 I-cache misses are very rare. Three benchmarks (vpr, gcc and mcf) have a significant fraction of branch mispredictions that follow an L2 D-cache miss by more than W instructions. All these interactions have the same impact on the branch misprediction penalty—the interval length is the only factor that affects the interaction, *i.e.*, the longer the interval length, the larger the branch misprediction penalty. Branch mispredictions that follow an L2 D-cache miss within ROB-size instructions result in a different penalty. A branch misprediction that is independent of the L2 D-cache miss is hidden under the L2 miss. This seems to be the case for gap (7.5%), vpr (5.2%), twolf (3.5%), etc. A branch misprediction that depends on an L2 D-cache miss sees a penalty that is nearly the same as the front-end pipeline length. This is a substantial fraction of the total number of branch mispredictions for a couple of benchmarks: 56.9% (mcf), 11.9% (vpr), 8.3% (gcc), etc.

7 Impact of miss event burstiness on branch misprediction penalty

One of the main conclusions from the previous sections is that the branch misprediction penalty is highly affected by the length of the inter-miss intervals. In other words, miss event burstiness affects the branch misprediction penalty. When talking about miss event burstiness, we are not just refering to branch mispredictions, but to the collective burstiness of miss events. Miss event burstiness is affected by the branch predictor implementation and the cache organization, *i.e.*, different miss rates lead to different miss event clustering, or even the same miss rate may lead to different miss event clustering. This section studies in more detail the interaction between miss event burstiness and the branch misprediction penalty. We first focus on the impact of the branch predictor implementation on the branch misprediction penalty and then consider the impact of the cache organization on the branch misprediction penalty.

7.1 Effect of branch predictor implementation

Thus far, we have been simulating one particular branch predictor, namely the one given in Table 2. However, the branch misprediction penalty is not independent of the specific branches that are mispredicted. A branch predictor with burstier miss behavior may suffer a different average branch misprediction penalty than another branch predictor (with the same misprediction rate) that shows less bursty miss behavior. Consequently, there is a relationship between the average penalty and the branch predictor that is used.

To quantify this effect, we measured the branch misprediction penalty and the branch misprediction rate for four different branch predictors—we used a collection of hybrid (bimodal+gshare) and bimodal predictors each resulting in a different misprediction rate. The results are shown in Figure 15 for the various benchmarks; the multiple dots per benchmark show the four branch predictors. There are several interesting observations to be made from this graph. First, the branch misprediction penalty for a given program generally increases with better predictors that decrease the miss rate. This is to be expected as a poorer predictor (resulting in a higher miss rate) will generally increase the burstiness of the branch mispredictions. This is not always the case, however; for example for twolf, there is the case where the misprediction penalty is (slightly) higher for a higher miss rate. Second, it is also interesting to observe that different benchmarks with the same misprediction rate can see different misprediction penalties. For example, when looking at a misprediction rate around 5%, mcf seems to have the highest misprediction penalty due to its poor L1 D-cache behavior, followed by vpr due to it low inherent ILP, followed by gzip and parser due to their sensitivity to the functional unit mix, etc. Similarly, around a mispredic-

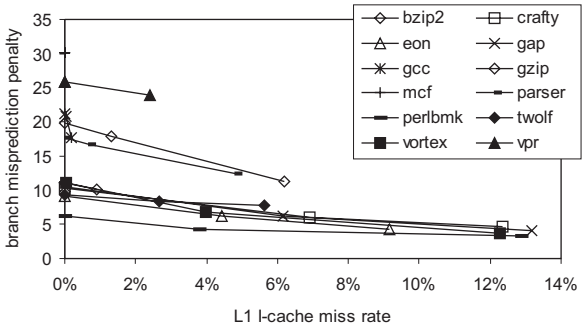

Figure 16. The branch misprediction penalty as a function of the L1 I-cache miss rate.

tion rate 20% and 25%, twolf seems to suffer from a larger branch misprediction rate than the other benchmarks. The reason is its low ILP, see Figure 10.

7.2 Branch misprediction penalty versus cache miss rates

Similar effects are to be expected when branch mispredictions are interleaved with bursty behavior due to other types of miss events. Figure 16 shows the relation between the branch misprediction penalty as a function of the L1 I-cache miss rate—we obtained similar results when varying the L2 cache (not shown here because of space constraints). The different miss rate numbers were obtained from different cache configurations. The L1 I-cache was varied from 8KB, 16KB to 32KB. As expected, we observe that the branch misprediction penalty decreases with increasing cache miss rates. The more bursty miss behavior, the shorter the inter-miss intervals and thus the shorter the branch misprediction penalty.

8 Summary and conclusions

Branch mispredictions are an important factor in determining superscalar processor performance. In this paper we studied the branch misprediction penalty, *i.e.*, the number of cycles lost per mispredicted branch. We showed that the branch misprediction penalty has two major contributors other than the front-end pipeline length. First, the branch misprediction penalty is dependent on the burstiness of the miss events, *i.e.*, the interval length or the number of instructions between two miss events. We conclude that the branch misprediction penalty increases with increasing interval length. As such, for the same number of branch mispredictions, programs with more non-branch miss events tend to have lower branch misprediction penalties. Second, the branch misprediction penalty is dependent on the average critical dependence path which is a function of the inherent ILP of a program, the functional unit mix and the number of short (L1) D-cache misses. Indeed, the branch misprediction penalty negatively correlates with the amount of ILP, *i.e.*, the lower the ILP, the higher the branch misprediction penalty. Similarly, for the same number of total miss events, programs with a higher fraction of L1 D-cache misses tend to have higher branch misprediction penalties. In addition, by constructing a stacked penalty model we are able to attribute fractions of the total penalty to specific program characteristics on a per-benchmark basis. For example, for vpr we observed that a major fraction of the total branch misprediction penalty is due its low ILP whereas for mcf a major fraction is due to its large number of L1 D-cache misses.

Acknowledgements

Stijn Eyerman and Lieven Eeckhout are supported by the Fund for Scientific Research—Flanders (Belgium) (FWO—Vlaanderen). This research is also supported by Ghent University, the HiPEAC Network of Excellence and the European SCALA project No. 27648. James E. Smith is supported by NSF grant CCR-0311361 and funds from the Intel Corporation.

References

[1] T. Karkhanis and J. E. Smith. A day in the life of a data cache miss. In *Proceedings of the 2nd Annual Workshop on Memory Performance Issues (WMPI 2002) held in conjunction with ISCA-29*, May 2002.

[2] T. S. Karkhanis and J. E. Smith. A first-order superscalar processor model. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA-31)*, pages 338–349, June 2004.

[3] P. Michaud, A. Seznec, and S. Jourdan. Exploring instruction-fetch bandwidth requirement in wide-issue superscalar processors. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT-1999)*, pages 2–10, Oct. 1999.

[4] P. Michaud, A. Seznec, and S. Jourdan. An exploration of instruction fetch requirement in out-of-order superscalar processors. *Internal Journal on Parallel Programming*, 29(1), Feb. 2001.

[5] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, pages 45–57, Oct. 2002.

[6] T. M. Taha and D. S. Wills. An instruction throughput model of superscalar processors. In *Proceedings of the 14th IEEE International Workshop on Rapid System Prototyping (RSP)*, June 2003.

[7] D. W. Wall. Limits of instruction-level parallelism. In *Proceedings of the fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pages 176–188, April 1991.